Udis86 Disassembler Library for x86 and x86-64

Documentation

Vivek Thampi

2008

Table of Contents

Using the library - libudis86

libudis86 can be used in a variety of situations, and the extent to which you need to know the API depends on the functionality you are looking for. At its core, libudis86 is a disassembler engine, which when given an input stream of machine code bytes, disassembles them for you to inspect. You could use it, to simply generate assembly language output of the code, or to inspect individual instructions and their operands.

Compiling and installing

libudis86 is developed for Unix-like environments, and the set of steps towards installing it on your system is very simple. Get the source tarball, unpack it, and,

```
$ ./configure
$ make
$ make install
```

is all you need to do. Of course, you may need to have root privileges to do a make install. The install scripts copy the necessary header and library files to appropriate locations on your system.

Interfacing with your program

Once you have installed libudis86, to use it with your program, first, include in your program the udis86.h header file,

```
#include <udis86.h>
```

and then, add the following flag to your GCC command-line options.

```
-ludis86
```

A Quick and Dirty Example

The following code is an example of a program that interfaces with *libudis86* and uses the API to generate assembly language output for 64-bit code, input from STDIN.

```
/* simple_example.c */

#include <stdio.h>
#include <udis86.h>

int main()
{
      ud_t ud_obj;

      ud_init(&ud_obj);
      ud_set_input_file(&ud_obj, stdin);
      ud_set_mode(&ud_obj, 64);
      ud_set_syntax(&ud_obj, UD_SYN_INTEL);

      while (ud_disassemble(&ud_obj)) {
            printf("\t%s\n", ud_insn_asm(&ud_obj));
      }

      return 0;
}
```

To compile the program (using gcc):

```
$ gcc -ludis86 simple_example.c -o simple_example
```

This example should give you an idea of how this library can be used. The following sections describe, in detail, the complete API of *libudis86*.

The API

The Udis86 Object

To maintain reentrancy and thread safety, udis86 does not use static data. All data related to the disassembly process are stored in a single object, called the udis86 object. So, to use libudis86 you must create an instance of this object,

```
ud_t my_ud_obj;
```

and initialize it,

```
ud_init(&my_ud_obj);
```

Of course, you can create multiple instances of libudis86 and spawn multiple threads of disassembly. That's entirely up to how you want to use the library. libudis86 guarantees reentrancy and thread safety.

The Functions

All functions in libudis86 take a pointer to the udis86 object (ud_t) as the first argument. The following is a list of all functions available.

`void ud_init (ud_t* ud_obj)`

ud_t object initializer. This function must be called on a udis86 object before it can used anywhere else.

`void ud_set_input_hook(ud_t* ud_obj, int (*hook)())`

This function sets the input source for the library. To retrieve each byte in the stream, libudis86 calls back the function pointed to by "hook". The hook function, defined by the user code, must return a single byte of code each time it is called. To signal end-of-input, it must return the constant, **`UD_EOI`**.

`void ud_set_user_opaque_data(ud_t* ud_obj, void* opaque);`

This function associates a pointer with the udis86 object to be retrieved and used in user functions, such as the input hook callback function.

`void* ud_get_user_opaque_data(ud_t* ud_obj);`

This function returns any pointer associated with the udis86 object, using the `ud_set_opaque_data` function.

`void ud_set_input_buffer(ud_t* ud_obj, unsigned char* buffer, size_t size);`

This function sets the input source for the library to a buffer of fixed size.

`void ud_set_input_file(ud_t* ud_obj, FILE* filep);`

This function sets the input source for the library to a file pointed to by the passed FILE pointer. Note that the library does not perform any checks, assuming the file pointer to be properly initialized.

`void ud_set_mode(ud_t* ud_obj, uint8_t mode_bits);`

Sets the mode of disassembly. Possible values are 16, 32, and 64. By default, the library works in 32bit mode.

`void ud_set_pc(ud_t*, uint64_t pc);`

Sets the program counter (EIP/RIP). This changes the offset of the assembly output generated, with direct effect on branch instructions.

`void ud_set_syntax(ud_t*, void (*translator)(ud_t*));`

libudis86 disassembles one instruction at a time into an intermediate form that lets you inspect the instruction and its various aspects individually. But to generate the assembly language output, this intermediate form must be translated. This function sets the translator. There are two inbuilt translators,

`UD_SYN_INTEL` - for INTEL (NASM-like) syntax.
`UD_SYN_ATT` - for AT&T (GAS-like) syntax.

If you do not want libudis86 to translate, you can pass a NULL to the function, with no more translations thereafter. This is particularly useful for cases when you only want to identify chunks of code and then create the assembly output if needed.

If you want to create your own translator, you must pass a pointer to function that accepts a pointer to ud_t. This function will be called by libudis86 after each instruction is decoded.

```
void ud_set_vendor(ud_t*, unsigned vendor);
```

Sets the vendor of whose instruction to choose from. This is only useful for selecting the VMX or SVM instruction sets at which point INTEL and AMD have diverged significantly. At a later stage, support for a more granular selection of instruction sets maybe added.

UD_VENDOR_INTEL - for INTEL instruction set.
UD_VEDNOR_ATT - for AMD instruction set.

```
unsigned int ud_disassemble(ud_t*);
```

This function disassembles the next instruction in the input stream. RETURNS, the number of bytes disassembled. A 0 indicates end of input. NOTE, to restart disassembly, after the end of input, you must call one of the input setting functions with the new input source.

```
unsigned int ud_insn_len(ud_t* u);
```

Returns the number of bytes disassembled.

```
uint64_t ud_insn_off(ud_t*);
```

Returns the starting offset of the disassembled instruction relative to the program counter value specified initially.

```
char* ud_insn_hex(ud_t*);
```

Returns pointer to character string holding the hexadecimal representation of the disassembled bytes.

```
uint8_t* ud_insn_ptr(ud_t* u);
```

Returns pointer to the buffer holding the instruction bytes. Use ud_insn_len(), to determine the length of this buffer.

char* ud_insn_asm(ud_t* u);

If the syntax is specified, returns pointer to the character string holding assembly language representation of the disassembled instruction.

void ud_input_skip(ud_t*, size_t n);

Skips n number of bytes in the input stream.

Examining an Instruction

After calling ud_disassembly, instructions can be examined by accessing fields of the ud_t object, as described below.

ud_mnemonic_code_t ud_obj->mnemonic

The mnemonic code for the disassembled instruction. All codes are prefixed by **UD_I**, such as, **UD_Imov**, **UD_Icall**, **UD_Ijmp**, etc.

ud_operand_t ud_obj->operand[n]

The array of operands of the disassembled instruction. A maximum of three operands are allowed, indexed as 0, 1, and 2. Operands can be examined using the their sub-fields as described below.

ud_type_t ud_obj->operand[n].type

This field represents the type of the operand n. Possible values are -

- **UD_OP_MEM** - A Memory Addressing Operand.
- **UD_OP_REG** - A Register Operand.
- **UD_OP_PTR** - A Segment:Offset Pointer Operand.
- **UD_OP_IMM** - An Immediate Operand
- **UD_OP_JIMM** - An Immediate Operand for Branch Instructions.
- **UD_OP_CONST** - A Constant Value Operand.
- **UD_NONE** - No Operand.

ud_obj->operand[n].size

This field gives the size of operand n. Possible values are - 8, 16, 32, 48, 64.

ud_obj->operand[n].base
ud_obj->operand[n].index
ud_obj->operand[n].scale
ud_obj->operand[n].offset
ud_obj->operand[n].lval

For operands of type **UD_OP_MEM**,

- **ud_obj->operand[n].base** is the base register (if any),
- **ud_obj->operand[n].index** is the index register (if any),
- **ud_obj->operand[n].scale** is the scale (if any),
- **ud_obj->operand[n].offset** is the size of displacement/offset to be added (8,16,32,64),
- **ud_obj->operand[n].lval** is displacement/offset (if any).

For operands of type **UD_OP_REG**,

- **ud_obj->operand[n].base** field gives the register.

For operands of type **UD_OP_PTR**,

- **ud_obj->operand[n].lval** holds the segment:offset.
- **ud_obj->operand[n].size** can have two values 32 (for 16:16 seg:off) and 48 (for 16:32 seg:off).

For operands of type **UD_OP_IMM**, **UD_OP_JIMM**, **UD_OP_CONST**,

- **ud_obj->operand[n].lval** holds the value.

Possible values for **ud_obj->operand[n].base** and **ud_obj->operand[n].index**.

```
/* No register */
UD_NONE,

/* 8 bit GPRs */
```

```
UD_R_AL,    UD_R_CL,    UD_R_DL,    UD_R_BL,
UD_R_AH,    UD_R_CH,    UD_R_DH,    UD_R_BH,
UD_R_SPL,   UD_R_BPL,   UD_R_SIL,   UD_R_DIL,
UD_R_R8B,   UD_R_R9B,   UD_R_R10B,  UD_R_R11B,
UD_R_R12B,  UD_R_R13B,  UD_R_R14B,  UD_R_R15B,

/* 16 bit GPRs */
UD_R_AX,    UD_R_CX,    UD_R_DX,    UD_R_BX,
UD_R_SP,    UD_R_BP,    UD_R_SI,    UD_R_DI,
UD_R_R8W,   UD_R_R9W,   UD_R_R10W,  UD_R_R11W,
UD_R_R12W,  UD_R_R13W,  UD_R_R14W,  UD_R_R15W,

/* 32 bit GPRs */
UD_R_EAX,   UD_R_ECX,   UD_R_EDX,   UD_R_EBX,
UD_R_ESP,   UD_R_EBP,   UD_R_ESI,   UD_R_EDI,
UD_R_R8D,   UD_R_R9D,   UD_R_R10D,  UD_R_R11D,
UD_R_R12D,  UD_R_R13D,  UD_R_R14D,  UD_R_R15D,

/* 64 bit GPRs */
UD_R_RAX,   UD_R_RCX,   UD_R_RDX,   UD_R_RBX,
 UD_R_RSP,  UD_R_RBP,   UD_R_RSI,   UD_R_RDI,
UD_R_R8,    UD_R_R9,    UD_R_R10,   UD_R_R11,
UD_R_R12,   UD_R_R13,   UD_R_R14,   UD_R_R15,

/* segment registers */
UD_R_ES,    UD_R_CS,    UD_R_SS,    UD_R_DS,
UD_R_FS,    UD_R_GS,

/* control registers*/
UD_R_CR0,   UD_R_CR1,   UD_R_CR2,   UD_R_CR3,
UD_R_CR4,   UD_R_CR5,   UD_R_CR6,   UD_R_CR7,
UD_R_CR8,   UD_R_CR9,   UD_R_CR10,  UD_R_CR11,
UD_R_CR12,  UD_R_CR13,  UD_R_CR14,  UD_R_CR15,

/* debug registers */
UD_R_DR0,   UD_R_DR1,   UD_R_DR2,   UD_R_DR3,
UD_R_DR4,   UD_R_DR5,   UD_R_DR6,   UD_R_DR7,
UD_R_DR8,   UD_R_DR9,   UD_R_DR10,  UD_R_DR11,
UD_R_DR12,  UD_R_DR13,  UD_R_DR14,  UD_R_DR15,

/* mmx registers */
UD_R_MM0,   UD_R_MM1,   UD_R_MM2,   UD_R_MM3,
UD_R_MM4,   UD_R_MM5,   UD_R_MM6,   UD_R_MM7,
```

```
/* x87 registers */
UD_R_ST0,   UD_R_ST1,   UD_R_ST2,   UD_R_ST3,
UD_R_ST4,   UD_R_ST5,   UD_R_ST6,   UD_R_ST7,

/* extended multimedia registers */
UD_R_XMM0,  UD_R_XMM1,  UD_R_XMM2,  UD_R_XMM3,
UD_R_XMM4,  UD_R_XMM5,  UD_R_XMM6,  UD_R_XMM7,
UD_R_XMM8,  UD_R_XMM9,  UD_R_XMM10, UD_R_XMM11,
UD_R_XMM12, UD_R_XMM13, UD_R_XMM14, UD_R_XMM15,

UD_R_RIP
```

Possible values for **ud_obj->operand[n].lval** depend on **ud_obj->operand[n].size**, based on which you could use its sub-fields to access the integer values.

- **ud_obj->operand[n].lval.sbyte** - Signed Byte
- **ud_obj->operand[n].lval.ubyte** - Unsigned Byte
- **ud_obj->operand[n].lval.sword** - Signed Word
- **ud_obj->operand[n].lval.uword** - Unsigned Word
- **ud_obj->operand[n].lval.sdword** - Signed Double Word
- **ud_obj->operand[n].lval.udword** - Unsigned Double Word
- **ud_obj->operand[n].lval.sqword** - Signed Quad Word
- **ud_obj->operand[n].lval.uqword** - Unsigned Quad Word
- **ud_obj->operand[n].lval.ptr.seg** - Pointer Segment in Segment:Offset
- **ud_obj->operand[n].lval.ptr.off** - Pointer Offset in Segment:Offset

Prefix Fields - These fields store prefixes (if found). If a prefix does not exists then the field corresponding to it has the value **UD_NONE**.

- **ud_obj->pfx_rex** - 64-bit mode REX prefix
- **ud_obj->pfx_seg** - Segment register prefix
- **ud_obj->pfx_opr** - Operand-size prefix (66h)
- **ud_obj->pfx_adr** - Address-size prefix (67h)
- **ud_obj->pfx_lock** - Lock prefix
- **ud_obj->pfx_rep** - Rep prefix
- **ud_obj->pfx_repe** - Repe prefix
- **ud_obj->pfx_repne** - Repne prefix

Possible values for **ud_obj->operand[n].pfx_seg** are,

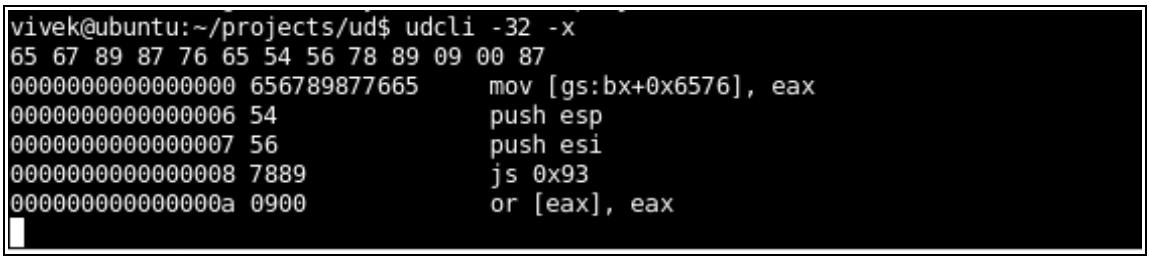
UD_R_ES, UD_R_CS, UD_R_SS, UD_R_DS,
UD_R_FS, UD_R_GS, UD_NONE

uint64_t ud_obj->pc

The program counter (EIP/RIP).

Using the command-line tool - udcli

A front-end incarnation of this library, udcli is a small command-line tool for your quick disassembly needs.



Usage

```
$ udcli [-option[s]] file
```

Options

```
-16      : Set the disassembly mode to 16 bits.
-32      : Set the disassembly mode to 32 bits. (default)
-64      : Set the disassembly mode to 64 bits.
-intel   : Set the output to INTEL (NASM like) syntax. (default)
-att     : Set the output to AT&T (GAS like) syntax.
-v <v>   : Set vendor. <v> = {intel, amd}
-o <pc>  : Set the value of program counter to. (default = 0)
-s <pc>  : Set the number of bytes to skip before disassembly to.
-c <pc>  : Set the number of bytes to disassemble to.
-x       : Set the input mode to whitespace seperated 8-bit numbers in
           hexadecimal representation. Example: 0f 01 ae 00
-noff    : Do not display the offset of instructions.
-nohex   : Do not display the hexadecimal code of instructions.
-h       : Display help message.
```

The hexadecimal input mode

Noteworthy among the command-line options of the udcli is "-x" which sets the input mode to whitespace separated 8-bit numbers in hexadecimal representation. This could come as a handy tool, for quickly disassembling hexadecimal representation of machine code, like those generated during software crashes, etc.